

A Guide to Creating Custom Triggers

-By Baseballkid14

For

Age of Mythology

And

Age of Mythology: The Titans Expansion

Table of Contents...

Section 1: Getting Started	4
1.1 – Tools You Will Need.....	4
1.2 – Setting Up Your XML.....	4
1.3 – Basic Triggering Rules.....	5
1.4 – Paramaters (Param).....	7
1.5 – Replacing Param Names With Custom Text/Values.....	8
1.6 – When to Add Quotes.....	10
1.7 – Using One Param for Two Commands.....	11
Section 2: Conditions	11
2.1 – Basic Rules.....	11
Section 3: Effects	12
3.1 – Basic Rules.....	12
Section 4: Expressions and Commands	12
4.1 – Expressions.....	12
4.2 – Commands.....	13
4.3 – kb Commands.....	13
4.4 – List of tr (Trigger) Commands.....	13
4.5 – List of kb Commands.....	16
Section 5: Extra Things That May Help	16
5.1 – Good Rules of Thumb.....	16
5.2 – Opening The triggertemp.xls (TT Users Only).....	16

Section 6: More Advanced (But Still Simple) Features.....17

6.1 – if, for, and else.....17

6.2 – The Loop Function.....19

6.3 – NOT, OR, and AND.....20

Section 7: Troubleshooting (“Why Doesn’t My Trigger work?”).....23

7.1 – XML Errors.....23

7.2 – Trigger Lock.....24

7.3 – Doesn’t Work Right.....,25

Section 1: Getting Started

1.1 – Tools You Will Need

In order to create custom triggers, you will need some text program that can save into other file types such as XML. **Notepad** is the easiest one to use and comes standard on all PCs. It should also be on most Macintosh computers. That's pretty much all you'll need to start making your own custom triggers! **WordPad** may also be used to make your own triggers but I prefer to use **Notepad**. Both of these programs can be found at: *Start Menu* → *(All) Programs* → *Accessories* on a standard PC. Another program you can use is **C++**. It's perfect for making XML files and it's free to download on Microsoft's official website. But for this tutorial, I am going to show how to make your own triggers in Notepad.

1.2 – Setting Up Your XML

Before you get started, it is best if you have some prior knowledge in XML programming and how it works. If not, don't worry. Before you get started, there are some basic rules you should know. The most reliable program you should use is **Notepad**. It is a free program that comes with most computers. You should get used to using notepad and get away from other text creators such as Microsoft Word.

The first of which is how trigger document should be set up:

```
<?xml version = "1.0"?>
```

```
<trigger version="2">
```

```
<Conditions>
```

```
</Conditions>
```

```
<Effects>
```

</Effects>

</trigger>

This is how your trigger (XML) document should be set up. Just copy and past this into Notepad or write it manually. The `<?xml version = "1.0"?>` needs to be included in all XML documents. You can also include addition information behind the "1.0" and before the ?. In the files contained within the data folder, it includes `encoding="UTF-8"` in between those two. When creating triggers, you **DO NOT** want to include this. `<trigger version="2">` needs to be included in trigger documents to let the game know it should be included in the trigger list. Any condition you make needs to be in after this tag: `<Conditions>` (A tag is term used for something within < and > in an XML document), but before this tag: `</Conditions>`. The same works for Effects (Put effects between `<Effects>` and `</Effects>`). Then you need to end the XML document by placing `</trigger>` at the end. A good rule of thumb is that any tag created also has to be ended. A tag being created is something like: `<Tag>` (E.G. `<Effects>` or `<Conditions>`). A tag being ended looks something like: `</Tag>` (E.G. `</Effects>` or `</Conditions>`). It has to contain / (slash) in immediately after the <. The tag **MUST** be ended with the same one it started as (E.G. you cannot end `<Effects>` with `</Conditions>`). If you do, your XML will break down and your triggers won't show up in the editor. I will talk more about this in *Section 7*.

1.3 – Basic Triggering Rules

There are many rules you need to follow in order for your trigger to work properly. In the **Command**, each value that changes the outcome is separated by commas (,). If you don't have the correct amount of commas specified to each command, your triggers won't work. The values must also be in the right order or it will not work properly, but the triggers will not cause all your triggers to stop working in some, but not all cases. You can, however change it so it does nothing in most cases. In these next steps, I will guide of how to change this so it does nothing.

```

<Effect name="$$22484$$Counter:Add Timer">
  <Param name="Name" dispName="$$22365$$Name" varType="string">countdown</Param>
  <Param name="Start" dispName="$$22485$$Start" varType="long">30</Param>
  <Param name="Stop" dispName="$$20926$$Stop" varType="long">0</Param>
  <Param name="Msg" dispName="$$20056$$Message" varType="stringid">The End is Near</Param>
  <Param name="Event" dispName="$$22362$$Trigger" varType="event">0</Param>
  <Command>trCounterAddTime("%Name%", %Start%, %Stop%, "%Msg%", %Event%);</Command>
</Effect>

```

This is the basic *Counter: Add Timer* Effect. Now you should notice that all the characters within the two Command tags (<Command and </Command>) are similar to the Param name's. You can change the name of any Param name as long as you do on in the Command as well. For Example:

```

<Param name="Name" dispName="$$22365$$Name" varType="string">countdown</Param>

```

I can change the Param name of this to any name I chose:

```

<Param name="Counter" dispName="$$22365$$Name" varType="string">countdown</Param>

```

But you also have to change it in the Command for it to still work:

```

<Command>trCounterAddTime("%Counter%", %Start%, %Stop%, "%Msg%", %Event%);</Command>

```

This is just another way of writing the *Counter: Add Timer* Effect that will accomplish the same thing. Although this is good to know, I have yet to show you how to make something do nothing. You need to substitute a value for the part of the command you don't want to do anything with whatever does nothing. In the case that you're dealing with a number, **-1** is the most common thing to use. And believe it or not, an event is a number to the trigger engine. It doesn't retrieve the trigger name that you see, it get's the trigger id. So we are just going to make it so it has a Name, a Start Time, and End Time, a Message, but no event is fired after the timer is complete. I realize this will not help anyone but it's good for understanding how these things work. I will now show you how to make no event fire:

```

<Command>trCounterAddTime("%Counter%", %Start%, %Stop%, "%Msg%", -1);</Command>

```

You probably noticed it's the same exact thing except instead of %Event%, it's -1. You now have no need for the Event param so you can just delete that line (From *<Param* to *</Param>*). There are things you should watch out for still:

```
<Command>trCounterAddTime("%Counter%", %Start%, %Stop%, "%Msg%", %-1%);</Command>
```

You don't want to this because it will look for a Param name called -1 (*<Param name="-1"...*), it will not fire no event.

```
<Command>trCounterAddTime("%Counter%", %Start%, %Stop%, "%Msg%");</Command>
```

Don't just delete this the -1 or %Event% because it now has 1 less comma which we talked about earlier is not a good thing. Here is what the final product should look like:

```
<Effect name="$$22484$$Counter:Add Timer">
  <Param name="Counter" dispName="$$22365$$Name" varType="string">countdown</Param>
  <Param name="Start" dispName="$$22485$$Start" varType="long">30</Param>
  <Param name="Stop" dispName="$$20926$$Stop" varType="long">0</Param>
  <Param name="Msg" dispName="$$20056$$Message" varType="stringid">The End is Near</Param>
  <Command>trCounterAddTime("%Counter%", %Start%, %Stop%, "%Msg%", -1);</Command>
</Effect>
```

You would usually see event in the editor, you now wont because that Param is gone. You may also substitute other values into that other than **-1** but I will get more in detail with that in Section 1.5.

If you change the dispName, you will also have to get rid of the **\$\$**'s and the numbers within them. This also applies to the **<Effect name** and the **<Condition name**. You can still get rid of them even if you don't change the name and nothing will become of to make it easier to read and interpret but not change what it does or looks like in the editor, all you have to do is change the normal code to:

```
<Effect name="Counter:Add Timer">
  <Param name="Counter" dispName="Name" varType="string">countdown</Param>
  <Param name="Start" dispName="Start" varType="long">30</Param>
  <Param name="Stop" dispName="Stop" varType="long">0</Param>
  <Param name="Msg" dispName="Message" varType="stringid">The End is Near</Param>
  <Command>trCounterAddTime("%Counter%", %Start%, %Stop%, "%Msg%", -1);</Command>
</Effect>
```

1.4 – Parameters

A parameter is what shows up in the editor in terms of places to type, or buttons. It allows the user to type in custom values which will be used by the trigger engine to determine what to do (E.G. If you have the trigger *Move to Unit*, with Attack Move on yes, the unit will attack but if it's on no, the unit will just go to the other unit without attacking. This is just one example of how Parameters work.) You should never make more than 7 Params because it will mess up the trigger screen (if you try, you will see what I mean), 7 is the maximum amount you should have.) You define all the traits in a Parameter on a line set up somewhat like this:

```
<Param name="Player" dispName="$$22301$$Player" varType="player">0</Param>
```

<Param defines it as a parameter so it can be used later in Commands. **name="Player"** gives it a name the Command has to use in order for it to be affected. The name is within percent signs (%Player%). If the name was Money, it should look like **%Money%** in the command. The param name can be anything you want it to be but you can't have two params with the same name. **dispName="\$\$22301\$\$Player"** is what it looks like in the editor. In the editor, you'll notice the \$\$22301\$\$ doesn't show up. That is the number id that is located in the **(xpack)language.dll** of what it will say in the editor. If someone has a Spanish game installed, it will look different in the .dll and in the editor as a result. You cannot add a random number in there but you can delete it altogether so it only says **dispName="Player"** and it will look no different in the editor. **varType="player"** determines what kind of button, list, place to type, etc. will be. In this case it will be a dropdown list of players from 1-10(it is a bug that it doesn't include 11 and 12). Here is a list of all the **VarTypes** there are:

string – a place to type any characters

stringid – same as string

long – a place to type in numbers

unit – select a physical unit that is on the map

operator – a dropdown list containing: >, >=, ==, <=, and <

float – a place to type in numbers but decimal points can be used

area – a place to select on the map (a location)

player – a dropdown list of players 0-10(0 = gaia)

tech – a dropdown list of all the techs

protounit – a dropdown list of protounits

resource – a dropdown list of food, wood, and gold

alliance – a dropdown list of ally, neutral, and enemy

group – a dropdown list of armies

bool – an OnOff switch

kbstat – a dropdown list of stats (E.G. tribute sent)

god power – a dropdown list of god powers

camerainfo – a place that captures an area on the screen but not a physical x,y,z position(a position marking how far left, high, and right a unit is from the corner of the map)

sound – a filename starting from the *sound* folder

event – a dropdown list of triggers

If you put a non-existent varType in it, that line will not be anything in the editor so be careful. And finally, `>0</Param>` is what the value is pre-set to in the editor so in this case, the player is automatically set to gaia until manually modified. It also ends the tag so the XML doesn't crash. You may also notice in some triggers it's ended like `/>` instead. This makes it start with nothing in the editor.

1.5 – Replacing Param Names With Custom Text/Values

After learning about Parameters in the last section you should know how they work pretty well. Depending on what the user puts in, the result will differ.

When making triggers you can manually substitute your own text/values into where the parameter will be. Here is an example of what I mean:

```
<Effect name="Ally">
  <Param name="Player1" dispName="Player" varType="player">0</Param>
  <Param name="Player2" dispName="Is Allied To Player" varType="player">0</Param>
  <Command>trPlayerSetDiplomacy(%Player1%, %Player2%, "ally");</Command>
</Effect>
```

I changed the "%Status%" with "ally" so instead of letting the user manually chose. Some have quotes (") around the percent signs (%) while some do not. In most cases, if the varType is a string, it will have quotes but this does too. You if you change the VarType to a string, and it doesn't originally have quotes, it doesn't mean you should add them. Each command functions in a different way. For example:

```
<Command>trPlayerSetDiplomacy(%Player1%, %Player2%, "ally");</Command>
```

The ***trPlayerSetDiplomacy*** command needs 2 commas and three values/choices. It is set up like:

```
<Command>trPlayerSetDiplomacy( , , "");</Command>
```

You can add your own values in there, or add a Param by putting a the Param name in between two percent signs. For every ***tr*** Command included in the typetest.xml, you should know how it is set up by looking at it (E.G. you should know Army Deploy goes ("","", , , ,) because that is how it is set up.

1.6 – When To Add Quotes

Adding quotes when/where needed is a major part in getting your trigger to work. If you forget quotes, or add them where they shouldn't be, you're triggers will be likely to break down. If you're using c++, you don't even need to read this paragraph (although you still need to with the next) because it automatically add them for you. You need to add quotes after the following, always: ***Effect name=***, ***Param name=***, ***dispName=***, and ***VarType=***. Nothing good will become of it if you don't.

Here is the part that c++ ***doesn't*** automatically cover, in the commands and expressions. It is easiest to know this if you copy-past-change from other triggers

because all you have to do is edit the values. If the *%ParamNameHere%* has quotes around it, you keep them. If it doesn't, you don't add them as a good rule of thumb.

1.7 – Using one Param For Two Commands

To make it so you don't have to use more than one parameter that says "Player" or something, you can use the same parameter for more than one thing, as long as you want the value to be the same for both. For example:

```
<Effect name="Grant God Power And Tribute">
  <Param name="Player" dispName="Player" varType="player">0</Param>
  <Param name="PowerName" dispName="Power" varType="godpower">default</Param>
  <Param name="Count" dispName="Uses" varType="long">1</Param>
  <Param name="ResName" dispName="Resource To Tribute" varType="resource">food</Param>
  <Param name="Amount" dispName="Amount To Tribute" varType="long">100</Param>
  <Command>trTechGodPower(%Player%, "%PowerName%", %Count%);</Command>
<Command>trPlayerTribute(%Player%, "%ResName%", %Amount%, 0);</Command>
</Effect>
```

This grants a god power to and tributes from the same player with only 1 parameter. It's pretty simple to understand. The last part of the tribute command I switched with 0 so they would tribute to player 0 instead of a user-defined player. I went more in depth with this in section 1.5.

Section 2: Conditions

2.1 – Basic Rules

There are a few rules you have to follow when making conditions. First of all, for a condition to be a condition, it needs to start with **<Condition name=...** to classify it as a condition. Your code must also have to be between the **<Conditions>** tag and the **</Conditions>** end tag. Most everything about conditions are described in the section on parameters and the Expressions and Commands section.

Section 3: Effects

3.1 – Basic Rules

Effects are very similar to conditions. They are set up like `<Effect name=...` and must be ended with `</Effect>`. It also has to be between the `<Effects>` start tag and the `</Effects>` end tag for it to show up in the editor.

Section 4: Expressions And Commands

4.1 – Expressions

An *Expression* is the kind of tag used in conditions. An expression waits for it to return true before the Effect(s) are fired. You cannot use Effect commands in Expressions although you can use Condition commands in Effects. I will get more in detail with this in section 2.2. You may not have more than 1 Expression per condition but there are ways around this which I will greater explain in section 6.2. To start off, I will teach you how a basic condition works:

```
<Condition name="$22294$Distance to Unit">
  <Param name="SrcObject" dispName="$22295$Source Units" VarType="unit">default</Param>
  <Param name="DstObject" dispName="$22296$Target Unit" VarType="unit">default</Param>
  <Param name="Op" dispName="$22297$Operator" VarType="operator">=</Param>
  <Param name="Dist" dispName="$22298$Distance" VarType="float">0</Param>
<Command>trUnitSelectClear();</Command>
  <Command loop="" loopParm="SrcObject">trUnitSelect("%SrcObject%");</Command>
  <Expression>trUnitDistanceToUnit("%DstObject%") %Op% %Dist%</Expression>
</Condition>
```

This is the Condition of *Distance to Unit*. The Expression *trUnitDistanceToUnit* is calculates the distance the selected unit is to a given point. Outside of the parentheses, it has two user-defined parameters (*%Op%* and *%Distance%*). *%Op%* defines whether it the distance a unit is to a point is greater than (>), greater than or equal to (>=), equal to (==), less than or equal to (<=) or less than (<). The *%Dist%* is the amount compared to the distance a unit is from a point. This condition will activate the effects in the editor if/when the value returned by

the distance a unit is to another unit is whatever operator the user entered compared to the value of the distance entered.

4.2 – Commands

Commands act a little differently than Expressions. They are used mainly in effects but can also be used in Conditions. It tells the game what to do depending on what the Command is. *trPlayerGrantResources* is the command to grant a player a given amount of resources.

```
<Effect name="$$22457$$Grant Resources">
  <Param name="PlayerID" dispName="$$22301$$Player" varType="player">0</Param>
  <Param name="ResName" dispName="$$22455$$Resource" varType="resource">food</Param>
  <Param name="Amount" dispName="$$22456$$Amount" varType="long">100</Param>
<Command>trPlayerGrantResources(%PlayerID%, "%ResName%", %Amount%);</Command>
</Effect>
```

This effect grants a resource type depending on what was selected for the Param called *ResName*. It grants it to whoever the Param called *PlayerID* is and grants a sum of whatever the Param known as *Amount* is.

4.3 – kb Commands

There are 4 different kinds of commands included with ***Age of Mythology***. There are ai commands, tr commands, kb commands, and rm commands. ai commands stand for artificial intelligence , or computers that automatically play the game. tr commands are trigger commands used for triggers. I'm not really sure what kb commands stand for but they can be used when creating an AI, a random map, or a trigger. An rm command is used for making random maps. ai and rm commands are not compatible with triggers but kb are. They include a wide variety of possible commands that can be used. Some can get very confusing, however.

4.4 – List of tr (Trigger) Commands

trUnitSelectClear() – De-selects all units (This must be done before any units are selected)

trUnitSelect("%SrcObject%") – Selects a unit for use in a later command/expression.

trTime() – Returns the amount of time passed since the beginning of the game (in seconds).

trTimeMS() – Returns the amount of time passed since the beginning of the game (in milliseconds).

trUnitDistanceToUnit("%DstObject%") – Calculates the distance a selected unit is to another unit.

trCountUnitsInArea("%DstObject%",%Player%,"%UnitType%",%Dist%) – Counts the amount of units are within a given area around another unit.

trUnitDistanceToPoint(%DstPoint%) – Calculates the distance a selected unit is to a given point on the map.

trUnitAlive() – Checks to see if the selected unit is alive.

trUnitDead() – Checks to see if the selected unit is dead.

trUnitVisToPlayer() – Returns true if a Human player is has their camera looking on a unit, It has to be in their LOS but if it is in their LOS, it doesn't automatically fire, it waits until the player looks there.

trUnitHasLOS(%PlayerID%) – Returns true if the selected unit is within the LOS of the player.

trTechStatusActive(%PlayerID%, %TechID%) – Checks to see if the tech is available by a given player.

trTechStatusResearching(%PlayerID%, %TechID%) – Returns true if the given player is researching a given tech.

trUnitPercentComplete() – Returns a value of how much the selected unit is completed (in percent).

trUnitPercentDamaged() – Returns a value in percents of how much a selected unit is damaged.

trPlayerUnitCount(%PlayerID%) – Returns the amount of units a player has.

trPlayerBuildingCount(%PlayerID%) – Returns the amount of buildings a player has.

trPlayerUnitAndBuildingCount(%PlayerID%) – Returns the amount of units and buildings combined a player has.

trPlayerCountBuildingInProgress(%PlayerID%, "%ProtoUnit%") – Counts the amount of buildings a player is building.

trPlayerResourceCount(%PlayerID%, "%Resource%") – Returns a value of the amount of a certain resource a player has.

trPlayerDefeated(%PlayerID%) – Returns true if the specified player has been defeated.

trPlayerAtPopCap(%PlayerID%) – Returns true if the player is active (the player has neither won nor lost.)

trPlayerGetPopulation(%PlayerID%) – Returns a value of the players population.

trUnitGetContained() – Counts how many units are garrisoned in the selected unit (defined on the previous two lines).

trUnitIsSelected() – Returns true if the selected unit is selected by the player (it only works in single player mode).

trUnitTypeIsSelected("%ProtoUnit%") – Returns true if any protounit is selected.

trPlayerGetDiplomacy(%Player1%, %Player2%) – Returns the diplomacy status that the first player is to the second player (Ally, Neutral, Enemy).

trUnitIsOwnedBy(%Player%) – Returns true if the selected unit is owned by the specified player.

trGetWorldDifficulty() Returns the difficulty of the game (0=easy, 1=moderate, 2=hard, 3=titan).

trCinematicAbort() – Returns true if spacebar or escape is pressed while the cinematic mod is on.

trIsGadgetVisible("%Gadget%") – Returns true if a gadget is visible to the player.

trUnitOnLush(%Lush%,%Player%) – Returns true if the selected unit is on the gaia lush of a specified player.

trUnitGetIsContained("%UnitType%") – Returns true if the selected unit is garrisoned in a protounit (E.G. Relic is in Temple)

trChatHistoryContains("%Text%", %PlayerID%) – Returns true if the chat history contains specified text from a given player.

trGetStatValue(%PlayerID%, %StatID%) – Returns the value of a stat (E.G. Enemy Units Killed).

And many others... all you have to do is open the *typetest.xml* and look on for th expression.

4.5 – List of kb Commands

All the kb commands can be found here:

C: → Program Files → Microsoft Games → Age of Mythology → Docs → aom ai script help file

Then all you have to do is scroll down to the bottom where all the kb commands are listed. Every kb command can be used in triggers. Once you master kb commands, your world in triggers will be vastly expanded in terms of possible triggers you can create.

Section 5: Extra Things That Might Help

5.1 – Good Rules of Thumb

Some good rules of thumb when creating triggers are:

Always close your tags (<**TAG**> with </**TAG**>!!!

Use Quotes after an = sign (Param name =...) but not in a Command/Expression.

Only make a few triggers at a time, then see if there are any errors before making more.

Don't use more than 1 <**Expression**> tag per conditions. Use none for effects.

Save as an XML when you're done (_____.xml). To open an xml, **Right Click → Open With → Notepad** or... open the xml by **Double Clicking, Right Click → View Source**.

5.2 – Opening The triggertempt.xs (TT Users Only)

Opening the triggertemp.xs file may or may not be helpful to you. It is located at: Start Menu → (My) Documents → My Games → Age of Mythology → Trigger2 → triggertemp.xs. You can open it by opening a blank notepad document, navigating yourself to the file, selecting **All Files (*.*)** and clicking on the xs file. It shows how all your triggers work in the map called ~testing which is automatically saved when you playtest a scenario. It will show exactly what the Effect (Command) or Condition (Expression) does with already filled in parameters so there are no % signs anymore. It may not be helpful to you but it's there if you ever need it.

Section 6: More Advanced (But Still Simple) Features

6.1 – if, for, else, and break

The commands **if**, **for**, and **else** can be very useful when creating triggers. I will first explain how to use the **if** Command. It is used in a command tag (E.G. <Command>if...</Command>) to see if a condition is true at that given point in time, if it is, the effects after it will activate, if it isn't, the effects after it will **NOT** activate. Here is an example:

```
<Command>if(trPlayerResourceCount1, "Gold") > 50)trChatSendSpoofer(1, "Player 1 has more than 50 Gold");</Command>
```

This line of code would send chat saying player 1 has more than 50 Gold if player 1 has more than 50 gold, if player 1 doesn't have at least 51 gold, nothing will happen. Unlike conditions, the if command does not wait for it to become true, it just checks to see if it's true at a given time, if it isn't but it does eventually become true, it will never activate **UNLESS** the trigger it is in is active and looped. This is where the *else* command comes into play. It will activate if that if command is not true so it would be set up something like:

```
<Command>if(trPlayerResourceCount1, "Gold") > 50)trChatSendSpoofer(1, "Player 1 has more than 50 Gold");</Command>
<Command>else</Command>
<Command>trChatSendSpoofer(1, "Player 1 doesn't have more than 50 Gold");</Command>
```

This effect will always send a chat but if the player has 51+ gold, it will say that, if he doesn't have 51+ gold, it will say he doesn't have it. The final command is for, it is a very useful Command to lessen the work of many things. It was used in a short version of *Modify All Protounits*, and it is also used in *Player Range* triggers. For sets a variable to more than one number. It is set up like:

```
<Command>for(Variable=Number; >ALowerNumber)</Command>
```

The *Variable* can be any letter(s) you chose. The *Number* and *ALowerNumber* determine what the value of the variable is. If the *Number* is 10 and the *ALowerNumber* is 1, it will set the variable to all numbers in between there(1,2,3,4,5,6,7,8,9,10). This is helpful for using an effect on more than one person and lots of other stuff as well. Now that you know how to set a variable to a number, all you have to do is use it in another command like so:

```
<Effect name="Kill All God Powers For Everyone">
  <Command>for(p=12;>0)trPlayerKillAllGodPowers(p);</Command>
</Effect>
```

All you have to do is substitute it in like any other custom value and you're done. This method makes p players 0-12, or all the players in the game and then kills their god power so everyone loses their god powers. The only thing left now is the **break** command. It is a relatively simple command but isn't used as much as the others. The break command stops the variable (from **for**) from being reaching other numbers by stopping the loop if it the command says for it to. That is not very easy to understand because I am not good at explaining this very well. Perhaps an example will help you to understand it's nature and use in triggers:

```
<Effect name="$$22443$$Send Chat">
  <Param name="PlayerID" dispName="$$22444$$From Player" VarType="player">0</Param>
  <Param name="Message" dispName="$$20056$$Message" VarType="stringid">default</Param>
```

```
<Command>for(p=12;>0)if(trPlayerAtPopCap(p))break trChatSend(p, "%Message%");</Command>  
</Effect>
```

This will send a chat from 1 person who is at their max pop. That person starts in numerical order because of how the **for** command was set up. If player 1 isn't at pop cap, it will check if player 2 is. If he isn't, it will check for player 3, and so on. Once it finds someone that is at their pop cap, it will set p to that value. If the break wasn't there, then it would send chat from anyone and everyone that was at their pop cap.

6.2 – The Loop Function

The loop function is a relatively simple topic. It isn't like the trigger loop that makes the trigger repeat itself. Before you learn what it does, you should know how to use it. { and } are the symbols used for loop. { begins a loop while } ends a loop. If you begin a loop but never end it, **ALL TRIGGERS WILL BREAK** so be careful! This eliminates the use of {player....(#)} triggers because the game will read it differently. The loop function tells the game to use what is before the { for everything within them. This may seem confusing as it is hard to explain but it is a simple topic. An example would be:

```
<Command>for(t=12;>0) {</Command>  
<Command>trPlayerKillAllGodPowers(t); }</Command>
```

This uses the variable of t for everything within the loop. If you are familiar with the **for** Command, then you should know that it sets the variable of t to every number equal to or lower than 12 and equal to and higher than 0. This gives t many different answers... t is equal to 0,1,2,3,4,5,6,7,8,9,10,11, and 12 all at the same time. Some things to watch out for:

```
<Command>for(t=12;>0) }</Command>  
<Command>trPlayerKillAllGodPowers(t);</Command>
```

Or

```
<Command>for(t=12;>0)</Command>  
<Command>trPlayerKillAllGodPowers(t);</Command>
```

You shouldn't do this because t is only equal to all numbers 1-12 for everything within the loop, which in this case is nothing. It will just kill all the god powers of t which isn't assigned a number so it might cause trigger lock, or just nothing will happen.

```
<Command>for(t=12;>0) {</Command>  
<Command>trPlayerKillAllGodPowers(t);</Command>
```

```
<Command>for(t=12;>0)</Command>  
<Command>trPlayerKillAllGodPowers(t); }</Command>
```

This will cause trigger breakdown because the loop isn't properly made, in one it's not ended and in the other, it's never started. Always make sure you have the same amount of `{`'s as `}`'s in your trigger. Conditionals also work in this fashion. You use the **if** command and an expression then use the loop function to make everything within happen if the conditional is true. Everything else that is outside the loops will happen always and normally.

6.3 – NOT, OR, and AND

The **NOT**, **OR**, and **AND** commands can be very useful when creating conditions, and in some cases (that use the **if** function), effects as well. If you are familiar with the boxes in the editor that you can check, you should already understand what they are. If you don't then I'd suggest you learn by searching guides. The **NOT** command returns true if two parts are not equal. This may seem confusing at first. Say you have something like this:

```
<Expression>trPlayerResourceCount(%Player%, %ResName%) != %Count%</Expression>
```

This will generate the amount of a certain resource a player has, and if the player has any amount of resource other than the specified count, it will return true. The **OR** command returns true if any of the expressions are met. Finally, the **AND** is the default for conditions and returns true if **all** the expressions are true. Now that you know what each one is very basic, it's time you learn how to add them to triggers. The command for **NOT** is **!=**, the command for **OR** is **||** (shift-\), and the

command for **AND** is **&&**. We will operate on conditions for this tutorial, but works similar in effects containing **if**. I will first teach you how to use **NOT**. Like I said earlier, **NOT** is used by **!=**, but in some cases, you can switch **== true** with **== false** and it will accomplish the same thing. For the purpose of using **!=**, I am going to use an expression that **uses** a **==** (or **%Op%**) but **not** with a true at the end. Player Unit Count is a perfect example:

```
<Condition name="$$$22323$$Player Resource Count">
  <Param name="PlayerID" dispName="$$$22301$$Player" VarType="player">0</Param>
  <Param name="Resource" dispName="$$$22324$$Resource" VarType="resource">food</Param>
  <Param name="Op" dispName="$$$22297$$Operator" VarType="operator">==</Param>
  <Param name="Count" dispName="$$$22321$$Number" VarType="long">1</Param>
  <Expression>trPlayerResourceCount(%PlayerID%, "%Resource%") %Op% %Count%</Expression>
</Condition>
```

This uses an Operator (**%Op%** (Param #3)) so the not command works perfectly in this case. For the NOT to work properly, you have to switch an **==** operator with a **!=** operator. This returns true if the value is anything **but** equal, it can be greater, or less than. Here is what your trigger should now look like:

```
<Condition name="Player Resource Count (Is NOT)">
  <Param name="PlayerID" dispName="$$$22301$$Player" VarType="player">0</Param>
  <Param name="Resource" dispName="$$$22324$$Resource" VarType="resource">food</Param>
  <Param name="Count" dispName="$$$22321$$Number" VarType="long">1</Param>
  <Expression>trPlayerResourceCount(%PlayerID%, "%Resource%") != %Count%</Expression>
</Condition>
```

I replaced **%Op%** with **!=** and deleted the **"Op"** parameter because there is no longer any use of it. You can use this on any trigger that has an expression set up like:

```
<Expression>(Anything Can Go Here) %Op% (Anything Can Go Here)</Expression>
```

It can also work if **%Op%** is any operator (**>**,**=**,**==**,**<=**,**<**). Where **"(Anything Can Go Here)"** is, you can type in a value, or an expression which get's a value, such as the Player Resource Count. You can do one side with a value and another with an expression, or both with expressions. If both sides have a value, then you'd probably always know whether it will be met or not, rendering it un-useful.

Now I am going to show you how to use the **OR** command. It is represented by **||** between two expressions. Here is an example:

```
<Condition name="All Units Or Buildings Dead">
  <Param name="PlayerID" dispName="$22301$Player" VarType="player">0</Param>
  <Expression>trPlayerUnitCount(%PlayerID%)==0 || trPlayerBuildingCount(%PlayerID%)==0</Expression>
</Condition>
```

This will return true if either all the players units or all the players buildings are dead. This is a relatively simple subject so I see no need to further explain.

The final thing I have to cover on this topic is **AND**. It will fire if all the conditions (expressions) are true. It is represented by **&&**. But and works the same way as **OR**. The only difference is that the XML will break down when **&&** is inserted. There is a way around this however...

```
<Condition name="Player Resource Count2">
  <Param name="PlayerID" dispName="$22301$Player" VarType="player">0</Param>
  <Param name="Resource1" dispName="$22324$Resource" VarType="resource">food</Param>
  <Param name="Resource2" dispName="And" VarType="resource">wood</Param>
  <Param name="Op" dispName="$22297$Operator" VarType="operator">==</Param>
  <Param name="Count" dispName="$22321$Number" VarType="long">1</Param>
  <Expression>trPlayerResourceCount(%PlayerID%, "%Resource1%") %Op% %Count% &&
trPlayerResourceCount(%PlayerID%, "%Resource2%") %Op% %Count%</Expression>
</Condition>
```

Under normal circumstances, this would work if the player has enough of the "Resource1" Param and the "Resource2" Param. But because of the way XML files are set up, this would break the file. The **CDATA** code may be used to evade the error. **CDATA** is a code that tells the xml to read anything within it differently so it doesn't get errors, but if the code is incorrect, it wouldn't matter anyway. **CDATA** is set up like: **<![CDATA[INSERT CODE HERE]]>** where the *INSERT CODE HERE* would be where you type in the code that would have a XML-unfriendly character (such as the **&&**) The trigger should now look like this:

```

<Condition name="Player Resource Count2">
  <Param name="PlayerID" dispName="$2301$Player" VarType="player">0</Param>
  <Param name="Resource1" dispName="$2324$Resource" VarType="resource">food</Param>
  <Param name="Resource2" dispName="And" VarType="resource">wood</Param>
  <Param name="Op" dispName="$2297$Operator" VarType="operator">==</Param>
  <Param name="Count" dispName="$2321$Number" VarType="long">1</Param>
  <Expression><![CDATA[trPlayerResourceCount(%PlayerID%, "%Resource1%") %Op% %Count% &&
trPlayerResourceCount(%PlayerID%, "%Resource2%") %Op% %Count%</Expression>
]]></Condition>

```

This trigger is now usable in game!

Section 7: Troubleshooting (“Why Doesn’t My Trigger Work?”)

7.1 – XML Errors

In this section, I am going to explain all the XML errors I have gotten and explain how to avoid and fix them. There are many errors that can happen in an XML, but there is always a reason for it. Here is a list of errors and how to fix them:

End tag '(Name of Tag)' does not match the start tag '(Name of Tag)'. – This error occurs when your tags do not match. Chances of getting this error are slim when using ++. Say you have a trigger like this:

```

<Effect name="$22525$Set Player Won">
  <Param name="Player" dispName="$2301$Player" varType="player">0</Param>
  <Command>trSetPlayerWon(%Player%);</Command>

```

You forgot to end the XML with a **</Effect>** at the end of the trigger. Because you didn’t end the tag, the XML will break down. This will also occur if say you didn’t end one of the Param, Command, or Expression tags correctly. Capitalization *does* count and contributes to this error.

An invalid character was found in text content. – This means a character that is not supported by XML was found in your document. This can happen if you paste a quote from Microsoft office or another program that makes quotes

like: " or " instead of ". There are also other unsupported characters which it will show you under the error message in blue.

A string literal was expected, but no opening quote character was found. – This means you forgot to add a quote where it was expected by the XML parser. It should show you where you need to add them below it in blue.

Whitespace is not allowed at this location. – This means you probably added an **&** or an **<** do your document. It will show the character you used in blue just below it. To work around this (with **<**) just use **<** where you want the **<** to be. Or you can use the **CDATA** trick. It could also mean you put a space between the beginning of the tag and the core of the tag (E.G. instead of doing **<Effect>**, you did **< Effect >**).

If you have any other errors, you're on your own. Searching Google for the error is always effective. Another effective way to find the location of your error is by converting the Error-XML to XMB with AOMed which can be downloaded on AoM Heaven. When you convert it, it will give you the exact location of the error so you can easily fix it.

7.2 – Trigger Lock

If you experience a *trigger lock* (when all your triggers break as a result of 1 trigger), there is something wrong with your code, or it's just not meant to be right. One thing that causes it is starting a loop with a { but never ending it with a }. For every loop you make, it should also be ended. If you have 3 {s, you should have 3 }s, if you expect the trigger to function properly. Another thing that could cause it is replacing invalid values in the Command or forgetting quotes in the Command. If you have quotes there initially and you get rid of them, the trigger will break all others. If you enter an invalid value in place of a parameter, it will also break. For Example:

```
<Effect name="$$$22423$$Convert">  
  <Param name="SrcObject" dispName="$$$22421$$Unit" varType="unit">default</Param>
```

```

<Command>trUnitSelectClear();</Command>
  <Command loop="" loopParm="SrcObject">trUnitSelect("%SrcObject%");</Command>
  <Command>trUnitConvert(All Players);</Command>
</Effect>

```

This wouldn't work because where **All Players** (Some random text I thought up) is, it is meant to accept a number, not text. That is why this one would break all triggers. Unless you used the for command to set the variable "**All Players**" to a number, it wouldn't work.

Say you added quotes around a something that shouldn't have quotes, that would also destroy all your triggers. E.G.:

```

<Effect name="$$22423$$Convert">
  <Param name="SrcObject" dispName="$$22421$$Unit" varType="unit">default</Param>
  <Param name="PlayerID" dispName="$$22301$$Player" varType="player">0</Param>
  <Command>trUnitSelectClear();</Command>
  <Command loop="" loopParm="SrcObject">trUnitSelect("%SrcObject%");</Command>
  <Command>trUnitConvert("%PlayerID%");</Command>
</Effect>

```

That command isn't supposed to accept quotes and therefore will break down. If you don't add quotes where they are needed, that would also break the trigger.

Using an invalid tag would also cause all your triggers to stop working. Using a tag like **<Result>** would result in trigger breakdown because the scenario doesn't know how to read that tag. Also, not using correct capitalization could cause this (E.G. **<command>** instead of **<Command>**).

7.3 – Doesn't Work Right

If your trigger passes the XML and trigger lock with no errors, all should work right. In some occasions, however that is not the problem. Although the code is set up to be read by the xml correctly and the scenario as well, there really are many things that can cause them to work improperly. Sometimes, nothing happens, while other times, something happens completely wrong. If you experience this, I suggest looking the in the **triggertemt.xs** file (make sure you

test the map you want it on, or save your map as ***~testing***. Scroll down until you find your trigger and that might help you see what went wrong in it.